Quiz Today

Covers:

- Concepts from lab 3
- Material from lecture 3

Reading from:


Time:
Programming for Biologists and Biochemists

Lecture 4:
Functions, Modules, and Files

Brian Kidd
April 19, 2010
Overview

1. Housekeeping Items and Questions
2. Exception Handlers
3. Functions
4. Modules
5. Files
From Last Time

Posted Concept Clarification Document:

Check syllabus

Labs:

How are they going?

More complex (i.e. no inherent time limit now as you have a week to work on them)
Lab 3 Feedback

Overall this was a challenging lab

Codon finder (problem 1) was tricky

Dictionaries seem to be a sticking point

Loop concepts need some refreshment

Outline your code before*
Overview

1. Housekeeping Items and Questions
2. Exception Handlers
3. Functions
4. Modules
5. Files
Python Errors

In the process of executing Python code, everyone is going to run into some sort of error

Tracebacks: "trace back" through the code to display the preceding outputs and functions leading up to the error

Runtime errors: problem that occurs during the execution of the code
Examples of Runtime Errors

Wrong inputs for your code

Comparison of incompatible values

Wrong outputs from your code

NameError - name 'x' not defined
ValueError - list.index(x): x not in list
KeyError - 'b'
AttributeError - 'y' object has no attribute 'x'
TypeError - 'x' expected 1 arguments, got 0
IndexError - string index out of range
IOError - No such file or directory: ''
Exception Handling Statements

Adding statements to your code to handle "errors" (exceptions) to what is expected

try:
    statements
except ErrorClass:
    except-statements

Exception handling allows you to anticipate and recover from expected errors
Simple Exception Example

```
try:
    statements
except ErrorClass:
    except-statements

inSeq = raw_input("Please enter a sequence: ")
try:
    inSeq.index('M')
except ValueError:
    print 'M not found in %s' % inSeq
```
Overview

1. Housekeeping Items and Questions
2. Exception Handlers
3. Functions
4. Modules
5. Files
Modularity*

Concept of breaking up a complex system into smaller parts called "modules"

*Don't confuse this concept with the Python modules.
Modules*

Modules are functional units of code that have separable, or defined, boundaries

*Don't confuse this concept with the Python modules.
Modularity In Biology

Functional modules:

defined by specific tasks performed
Modularity In Biology

Example:

Visual Module*

*Crude breakdown of visual "module".
Modularity In Programming

Programs can be complex

Divide code into modules*

A module is a collection of logical statements that have been bundled as a single unit

*Don't confuse this concept with the Python modules.
Advantages of Modularity

Modules make code

- Easier to read and write
- Easier to reuse
- Easier to combine
Suppose you are asked to design a program that calculates a number of different physical and chemical characteristics (nucleic acid and protein):

- Count bases
- Count amino acids
- Calculate hydrophobic amino acids
- Calculate hydrophilic amino acids
- Calculate GC-richness of a sequence
- Calculate molecular weight, etc.

*Don't confuse this concept with the Python modules.*
Modularity Example

Count bases
Count amino acids
Calculate hydrophobic amino acids
Calculate hydrophilic amino acids
Calculate GC-richness of a sequence
Calculate molecular weight, etc.

Easier to Read/Write

Easier to Reuse

Easier to Combine
Functions: General Concept

Function is a "module*" within your code

Name the function (def)
Indentation code block
Pass input values to function – "call function"
Return results of function calculation

*Don't confuse this concept with the Python modules.
Functions: Code Within Code

Main Program

Function 1

Function 2

Function 3
Functions: Advantages

1. Allows you to reuse code
2. Easier to test your code
3. Organizes your code
4. More reliable/robust code
5. Speeds up development time
6. Break up into smaller problems
7. Only have to know inputs and outputs
Functions: Definition I

A function consists of:

- `def` name
- input parameters (zero or more)
- docstrings (optional information)
- set of statements in a code block (indentation)
- return value(s)*

```python
def addTwo(x, y):
    """adds two values together""
    s = x + y
    return s
```

*If return is not used explicitly, then the function returns `None` by default.*
A function consists of:

- `def`
- `name`
- input parameters (zero or more)
- docstrings (optional information)
- set of statements in a code block (indentation)
- return value(s)*

*Pass statement does nothing and can be used as a placeholder when developing your program.
Functions: Usage

Example 1:
Calculate distance between two atoms

Example 2:
DNA base validation

Example 3:
Calculate molecular weight
#!/usr/bin/env python
#
# Name: Brian Kidd (bkidd)
# Group: everyone in class
# Desc: Calculate distance between two atoms
#
# import math module for sqrt()
import math

# read in two sets of atomic coordinates, cast them as tuples and assign
# them a name. for information on tuples, see pages 61-62 in Model.
a1 = tuple(input("Enter the 1st set of coordinates separated by commas: "))
a2 = tuple(input("Enter the 2nd set of coordinates separated by commas: "))

# calculate the displacement of each coordinate
xDiff = a1[0] - a2[0]
yDiff = a1[1] - a2[1]

# calculate the distance using the multiplication operation and the sqrt() from
# the math library. note use of math.sqrt() to access the function.
dist = math.sqrt(xDiff*xDiff + yDiff*yDiff + zDiff*zDiff)

# print out the formatted output of distance
print "dist = %0.3f" % dist
Function to Calculate Distance

#!/usr/bin/env python
#
# Name: Brian Kidd (bkidd)
# Group: everyone in class
# Desc: Function to calculate distance between two atoms
Example 1: Distance Calculation

def calcDist (a1, a2):
    d2 = []
    for i in range(len(a1)):
        d2.append((a1[i] - a2[i])**2)
    return math.sqrt(math.fsum(d2))

name
input parameters (zero or more)
set of statements in code block
return value(s)

Assumptions with this function?
#!/usr/bin/env python
#
# Name: Brian Kidd (bkidd)
# Group: everyone in class
# Desc: Validate that a string contains only DNA bases
#
# set of DNA sequence type, which will be used to test the input sequence.
dnaSet = set('ACGT')

# read in string and convert case to all upper (page 57 in Model)
seq = raw_input("Enter sequence: ").upper()

# define Boolean flag that we'll use in a test condition later. the state of
# the flag is determined by whether the characters of the input sequence are a
# subset of the dnaSet.
isDNA = set(seq) <= dnaSet

# perform conditional statement to take the appropriate action on seq.
if isDNA:
    print "your sequence is DNA"
else:
    print "i don't recognize that sequence"
Function to Validate DNA

#!/usr/bin/env python
#
# Name: Brian Kidd (bkidd)
# Group: everyone in class
# Desc: Function to validate whether a sequence is DNA.
Example 2: Base Validation

```python
def isDNA(seq):
    return set(seq.upper()) <= set('ACGT')

def isDNA2(inSeq):
    seq = inSeq.upper()
    return len(seq) == (seq.count('T') + seq.count('C') + seq.count('A') + seq.count('G'))
```

name, input parameters(s), code block, return value(s)

Assumptions with either function?
Example 3: Molecular Weight

#!/usr/bin/env python
#
# Name: Brian Kidd (bkidd)
# Group: everyone in class
# Desc: Calculate the molecular weight of a protein from its primary sequence
#
# dictionary of molecular weights for each amino acid
aaWeight = { "A":89.093,  "R":174.201, "N":132.118, "D":133.103, "C":121.158,
           "E":147.129, "Q":146.145, "G":75.067,  "H":155.155, "I":131.173,
           "L":131.173, "K":146.188, "M":149.211, "F":165.189, "P":115.131,
           "S":105.093, "T":119.119, "W":204.225, "Y":181.189, "V":117.146 }

# convert input sequence to uppercase to match dictionary
protSeq = raw_input("Enter your protein sequence: ").upper()

# variables to hold value of molecular weight
totalMW = 0

# variable for molecular weight of water
mwWater = 18.015

# loop through each amino acid in the input sequence and lookup its MW
for aa in protSeq:
    totalMW += aaWeight.get(aa, 0)    # use get to avoid error and return 0
    totalMW -= (mwWater*(len(protSeq)-1))  # subtract off the water weight

# print out results
print "The net weight is: %0.1f" % (totalMW)
Function to Calculate MW

#!/usr/bin/env python
#
# Name: Brian Kidd (bkidd)
# Group: everyone in class
# Desc: Calculate the molecular weight of a protein from its primary sequence
Example 3: Molecular Weight

```python
def calcMW (inSeq, totalMW = 0):
    aaWeight = {"A":89, "R":174, "N":132, 
                "D":133, "C":121, "E":147, "Q":146, 
                "G":75, "H":155, "I":131, "L":131, 
                "K":146, "M":149, "F":165, "P":115, 
                "S":105, "T":119, "W":204, "Y":181, 
                "V":117 }
    mwWater = 18
    seq = inSeq.upper()
    for aa in seq:
        totalMW += aaWeight.get(aa, 0)
        totalMW -= (mwWater*(len(seq)-1))
    return totalMW
```

name, input parameters(s), code block, return value(s)

Assumptions with this function?
Function Exercise
Example pseudocode*:

read in protein sequence
function to calculate extinction coefficient
function to calculate absorbance
print out molecular weight, extinction coefficient and absorbance to screen

*Can reuse previous functions.
#!/usr/bin/env python
#
# Name: Brian Kidd (bkidd)
# Group: everyone in class
# Desc: Create a group of functions to perform the tasks that will estimate the
# absorbance of a protein sequence in water.
Namespace

A name can mean different things in different context

The space in which a name is defined

*Figure 2-2 in Model.*
A name can mean different things in different context

```python
a = 0
def nsExample(a):
    b = 5
    while a < 3:
        print a
        a += 1
    print b
nsExample(a)
print "a equals %s" % a
print "b equals %s" % b
```

Names in different namespaces are unrelated

*✔ OK!*

*✘ Gives an error!*

*✘ Gives an error!*

*✔ OK!*
Function Namespaces

Functions get their own namespace

for example: 

```python
def plusplus (x):
    x += 1
    return x
```

```
y = 4
z = plusplus (y)
print "%s, %s" % (y, z)  # 4, 5 instead of 5, 5
```
Namespace in Functions

Namespace defines where a name exists

To determine a name's space, find the first containing block that it has been defined within.

It is defined in this block, plus all sub-blocks defined within that block.

A function define its own block.

Global namespaces include built-in functions, methods, and data types.
Example of Namespaces

Global versus local

```python
def addTwo (x, y):
    sum = x + y
    print "%s + %s = %s" % (x, y, sum)
    print dummyGlobal
    return sum

x = 2
y = 2
dummyGlobal = "hello"
sum = 5
total = addTwo (x, y)  # total = 4
print sum               # sum = 5
```
A function takes in zero or more inputs
input parameters can have "default" values

```python
def cutDNA (seq, RE="GAATTC", offset=1):
    """splits a sequence of DNA into pieces based on a
restriction enzyme (RE) match"""
    newSeq = []
tmpSeq = seq
matchPos = tmpSeq.find(RE)
while matchPos > 0:
    cutPos = matchPos + offset
    newSeq.append(tmpSeq[:cutPos])
tmpSeq = tmpSeq[cutPos:]
matchPos = tmpSeq.find(RE)
newSeq.append(tmpSeq)
return newSeq
```
Nested Functions

Single call

Call 2 from 1

Call 3 from 2, from 1

*Page 25 in Model.*
Function Summary

Perform a specific task – the more focused that task, the better (think single-minded)

Think in terms of reusable items

def, name, parameters (defaults), return
docstrings
pass (useful for pseudocode)
assert* and error handling
nested functions calls

Define their own namespace

Built-in, versus your own, versus available

*Pages 30-32 in Model.
Overview

1. Housekeeping Items and Questions
2. Exception Handlers
3. Functions
4. Modules
5. Files
Modules

Files containing optional types, functions, and methods that allow you to:

**Recycle** bundles of names and functions that you want to use often

**Import** a module is a file that you can refer to from other files

**Organize** your code so it's easier to read and write
Module: Import

To get access to a module's contents, use the *import* statement*

**Statement**

**Importing**

The basic form of the import statement loads a module into the Python environment and makes its name available in the namespace into which it was imported (usually the interpreter or a program).

```python
import name
```

The *name* is just the name of the module—no path and no extension.

You've already seen this in lab 2:

```python
import math
```

*Page 34 in Model.*
Some Useful Modules

**os:** collection of > 150 functions and > 50 different data definitions specific to the operating system you're running on

http://docs.python.org/library/os

**math:** collection of 40 mathematical functions and 2 data definitions (e, pi)

http://docs.python.org/library/math

**sys:** collection of > 20 functions and > 40 different data definitions for interacting with the interpreter

http://docs.python.org/library/sys
You've already seen this in lab 2:

```python
import math

# calculate sqrt
math.sqrt(9)

# determine natural log – ln(x)
math.log(1)

# determine log base 2
math.log(8, 2)

# calculate sine of π/2
math.sin(math.pi/2)
```
Module: Selective Import

**Statement**

**Selective Import**

This form of `import` statement loads a library into the Python environment but does not make its name available in the namespace from which it was loaded. Instead, it imports specific names from the module into the importing namespace.

```python
from modulename import name1, name2, ...
```

You can give something you are importing a different name when you import it by using the following form of the `import` statement:

```python
from modulename import actualname as yourname
```

You can import *all* the names from a module with the following variation:

```python
from modulename import *
```

*Page 35 in Model.*

See pages 35-37 for a nice illustration of module namespace.
from math import sqrt, log

# calculate sqrt
sqrt(9)  # don't need math.

# determine natural log – ln(x)
log(1)  # don't need math.

# determine log base 2
log(8, 2)  # don't need math.

# calculate sine of π/2
math.sin(math.pi/2)  # produces an error (wasn't imported and math is not defined)
Create Your Own Module

Different programs that need access to the same functions or data

Examples:
converter.py from lab 2, seqTrans.py from lab 3
calculateMW.py and calculateEC.py from lab 3

What data do these programs share?

What should we do?
Including Our New Module

Write modules to reuse common data and functions whenever possible

**converter.py**

```python
#!/usr/bin/env python
import codonTables

print RNAcodonTable['AUG']
```

**seqTrans.py**

```python
#!/usr/bin/env python
import codonTables

print ThreeToOne[RNAcodonTable['AUG']]
```
Modules Summary

Collection of data and functions (mostly) that you bring into your program with import

After importing, all of the tools in the module are available to your program

Overall (total namespace) versus from (selected namespace)

```python
import mod1
from mod1 import func1, func2
from mod1 import func1 as newname
```

Allow you to reuse code, organizes your program

Built-in, versus your own, versus available
Overview

1. Housekeeping Items and Questions
2. Exception Handlers
3. Functions
4. Modules
5. Files
Streams

A stream is a *temporally ordered* sequence of *indefinite length* that travels from source ➔ sink.
Files as Streams of Data

**Read** data from a file into python

**Write** data from python to a file

*Page 74 in Model.*
Creating File Objects

In Python, files are processed using the built-in function \texttt{open}

Example:

\texttt{open (path, mode)}

- \texttt{path} is the location of the file
- \texttt{mode} is the content and operation performed on the file

\begin{table}
\centering
\caption{Content mode values for opening a file}
\begin{tabular}{|c|c|l|}
\hline
Value & Mode & Interpretation \\
\hline
\texttt{t} & Text (default) & Characters or strings \\
\texttt{b} & Binary & Bytes \\
\hline
\end{tabular}
\end{table}

\begin{table}
\centering
\caption{Use mode values for opening a file}
\begin{tabular}{|c|c|c|c|}
\hline
Value & Initial file position & Read? & Write? \\
\hline
\texttt{r} & Beginning (default) & Yes & No \\
\texttt{w} & Beginning & No & Yes \\
\texttt{a} & End & No & Yes \\
\texttt{r+} & Beginning & Yes & No \\
\texttt{w+} & Beginning & Yes & Yes \\
\texttt{a+} & End & Yes & Yes \\
\hline
\end{tabular}
\end{table}
Open Files Using **with**

Compound statement for opening and closing a file in Python

**STATEMENT**

**The with Statement**

The `with` statement is used to open and name a file, then automatically close the file regardless of whether an error occurs during the execution of its statements. Like a `def` statement, a `with` statement contains an indented block of statements.

```python
with open(path, mode) as name:
    statements-using-name
```

More than one file can be opened with the same `with` statement, as when reading from one and writing to the other.

```python
with open(path1, mode1) as name1, open(path2, mode2) as name2, ...:
    statements-using-names
```
File Handles

Open creates an object that is a "handle" to a file (*fileobj*).

Built-in methods for manipulating file objects.

**Reading:**

*fileobj.read()* - reads in binary file
*fileobj.readline()* - read single line of file
*fileobj.readlines()* - read all lines into list

**Writing:**

*fileobj.write(string)* - write string to *fileobj*
*fileobj.writelines(sequence)* - write each element of *sequence* to *fileobj*

*Page 76 in Model.*
File Methods

**Reading**

`fileobj.read([count])`
Reads `count` bytes, or until the end of the file, whichever comes first; if `count` is omitted, reads everything until the end of the file. If at the end of the file, returns an empty string. This method treats the file as an input stream of characters.

`fileobj.readline([count])`
Reads one line from the file object and returns the entire line, including the end-of-line character; if `count` is present, reads at most `count` characters. If at the end of the file, returns an empty string. This method treats the file as an input stream of lines.

`fileobj.readlines()`
Reads lines of a file object until the end of the file is reached and returns them as a list of strings; this method treats the file as an input stream of lines.

**Writing**

`fileobj.write(string)`
Writes `string` to `fileobj`, treating it as an output stream of characters.

`fileobj.writelines(sequence)`
Writes each element of `sequence`, which must all be strings, to `fileobj`, treating it as an output stream of lines. Note, however, that although this method’s name is intentionally analogous to `readlines`, newline characters are not added to the strings in `sequence` when they are written to `fileobj`.

*Page 76 in Model.*
Examples

Read in fasta file

Write out sequences to file in fasta format

Reminders:
use the os methods to help with the path*
  os.getcwd() - get current "working directory"
  os.chdir(path) - change to "path" as working dir
  os.mkdir(path) - make directory at "path"
remove last character of readline "\n" with .strip()
write will overwrite data

*See page 227-231 in Model for more methods.
Streams of data (read and write)

Mode values dictate:
  the content of data (text versus binary)
  how to access the data (read, write, append)

A file handle is an object that represents a file

Open file handles using the with function

Read through files:
  line by line – readline
  batch of lines – readlines

Write out data to files:
  line by line – write
  using a sequence of strings – writelines
Homework

Reading:
Model:
Chapters 7 & 8

Quiz:
Quiz 5 next Tues (4/26)
Covers readings, lectures, and lab

Project Deliverable:
Send me a two paragraph proposal
1. Introduce biological problem
2. Programming concepts for solving problem
Lab 4 Preview

Functions (1 program multiple parts)

1. Nucleic acid sequence analysis tool
   6 functions, two modules

Username/password: bme60/bme60

*remember to read the notes and hints document!*